May 24, 2019

REMOTE C++ DEVELOPMENT WITH DOCKER AND CLION (WITH X11)

My preferred Linux distribution of choice is Arch Linux, but many development SDKs that I need to use at work and at home are built for and tested for an Ubuntu environment. I could figure out how to get them to work in my native package manager, or I could just use Ubuntu (not happening), or use an Ubuntu VM (problematic for something like embedded development where I need solid USB access).

But there's a fourth option that I find preferable: run a Docker container for every project and connect the container to my IDE of choice (CLion), which will transparently transfer source files to the container, build them, and use remote GDB debugging.

A side benefit of this methodology is not cluttering your system with packages that you installed to build a specific thing. Sometimes you're experimenting and trying to get something to build, which requires you install dependency A, B, C, and D, but it turns out C actually wasn't necessary and a few months later you've forgotten what those packages were for or why you installed them. This keeps your host system clean, and you can do all of the experimentation in your container, and then delete it if you want.

Another benefit is reproducible builds. Related to the previous paragraph, you may install some dependencies to get something to build, forget that you installed them (or they were previously installed already), and not put them in the dependencies list of your project's README. Now anyone trying to build your project is left wondering what package they're missing.

- **Building a Base Docker Image**
- **Building a Project-Specific Docker Image**
- **A Simple Qt Application**

- **Configuring CLion**
- **Further Reading**

Building a Base Docker Image

We need a good base image that contains all of the necessary packages and configuration to be able to build with CMake, receive files over SSH, and be debugged remotely.

The **Dockerfile** looks like this:

```
1     FROM ubuntu:18.04
2
3     RUN apt update \
4         && apt upgrade -y \
5         && apt install -y \
6         apt-utils build-essential clang cmake gdb gdbse
7
8     # Taken from - https://docs.docker.com/engine/examp
9     RUN mkdir /var/run/sshd
10    RUN echo 'root:root' | chpasswd
11    RUN sed -i 's/PermitRootLogin prohibit-password/Per
12
13    # SSH login fix. Otherwise user is kicked off after
14    RUN sed 's@session\s*required\s*pam_loginuid.so@ses
15
16    ENV NOTVISIBLE "in users profile"
17    RUN echo "export VISIBLE=now" >> /etc/profile
18
19    # 22 for ssh server. 7777 for gdb server.
20    EXPOSE 22 7777
21
22    # Create dev user with password 'dev'
23    RUN useradd -ms /bin/bash dev
24    RUN echo 'dev:dev' | chpasswd
25
26    # Upon start, run ssh daemon
27    CMD ["/usr/sbin/sshd", "-D"]
```

You can build the image with **docker build**, supplying your Docker Hub username and repo name for a tag, if you want. For example:

```
docker build -t vertexmachina/docker-remote-dev .
```

I've already done this and uploaded the image to Docker Hub though, so you can instead just use my image as seen below.

Building a Project-Specific Docker Image

You'll likely need some extra packages for whatever you're developing. As an example, let's say you're building a Qt program, so you put together the following **Dockerfile**:

```
FROM vertexmachina/docker-remote-dev:latest

RUN apt update && apt install -y qt5-default
```

This uses the base image that we built before, and adds Ubuntu's default Qt package (with its numerous dependencies).

We'll use **docker-compose** to build an image specific for this project and set up everything. Here's the **docker-compose.yml** file:

```yaml
version: '3'

services:
  dev:
    build: # use the Dockerfile in the current dire
      context: .
      dockerfile: Dockerfile
    security_opt: # options needed for gdb debuggin
      - seccomp:unconfined
      - apparmor:unconfined
    container_name: dev
    ports:
      - "7776:22" # SSH
      - "7777:7777" # GDB Server
    volumes:
      - ${HOME}/.Xauthority:/home/dev/.Xauthority:r
      - /tmp/.X11-unix:/tmp/.X11-unix # X11 stuff
      - /dev/dri:/dev/dri #X11 stuff
      - /dev/snd:/dev/snd #X11 stuff
```

The **volumes** section is where the magic happens for connecting your host X server to the container.

Now you can spin it up with **docker-compose up -d**.

A Simple Qt Application

We'll create a very simple Qt application with a single file **Main.cpp**:

```
```

```cpp
#include <QApplication>
#include <QWidget>
#include <iostream>

int main(int argc, char __argv)
{
    QApplication app(argc, argv);

    QWidget widget;
    widget.setFixedSize(400, 400);

    QString helloString = "Hello from " + qgetenv("
    widget.setWindowTitle(helloString);

    widget.show();

    return QApplication::exec();
}
```

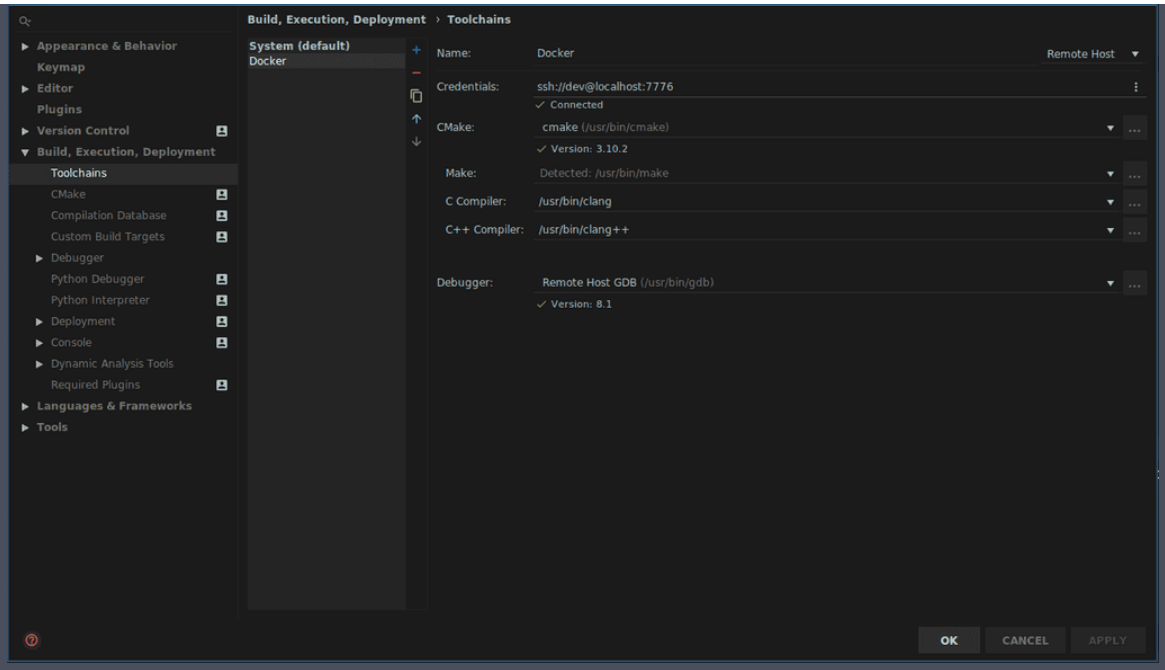
And the corresponding **CMakeLists.txt**:

```
cmake_minimum_required(VERSION 3.0)

project(hello)

find_package(Qt5 REQUIRED COMPONENTS Widgets)

add_executable(hello Main.cpp)

target_link_libraries(hello Qt5::Widgets)
```

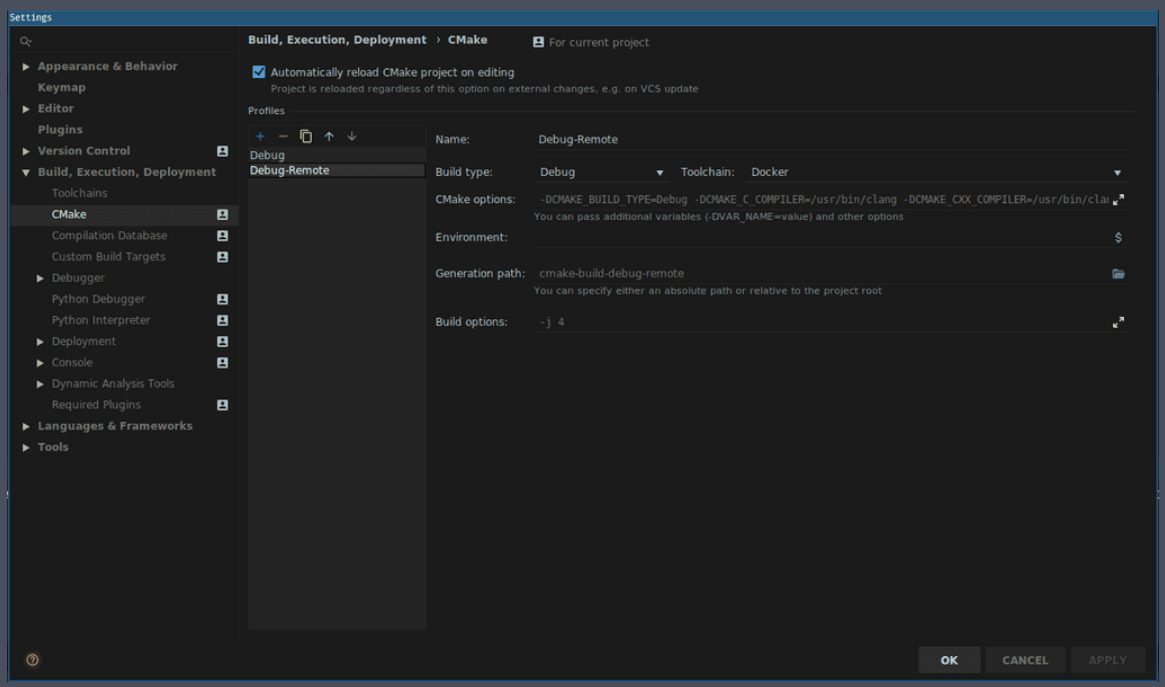
Configuring CLion

Start CLion and open the project directory containing the **CMakeLists.txt** and **Main.cpp**.

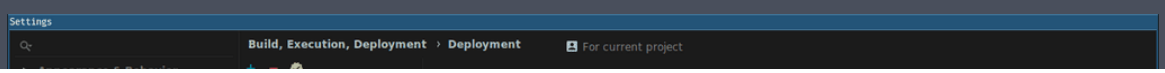
Go to **File -> Settings -> Build, Execution, Deployment -> Toolchains**. Add a new toolchain named **Docker**, set it to **Remote Host**, and configure the credentials using the three dots to the right of the field (username: dev password: dev).

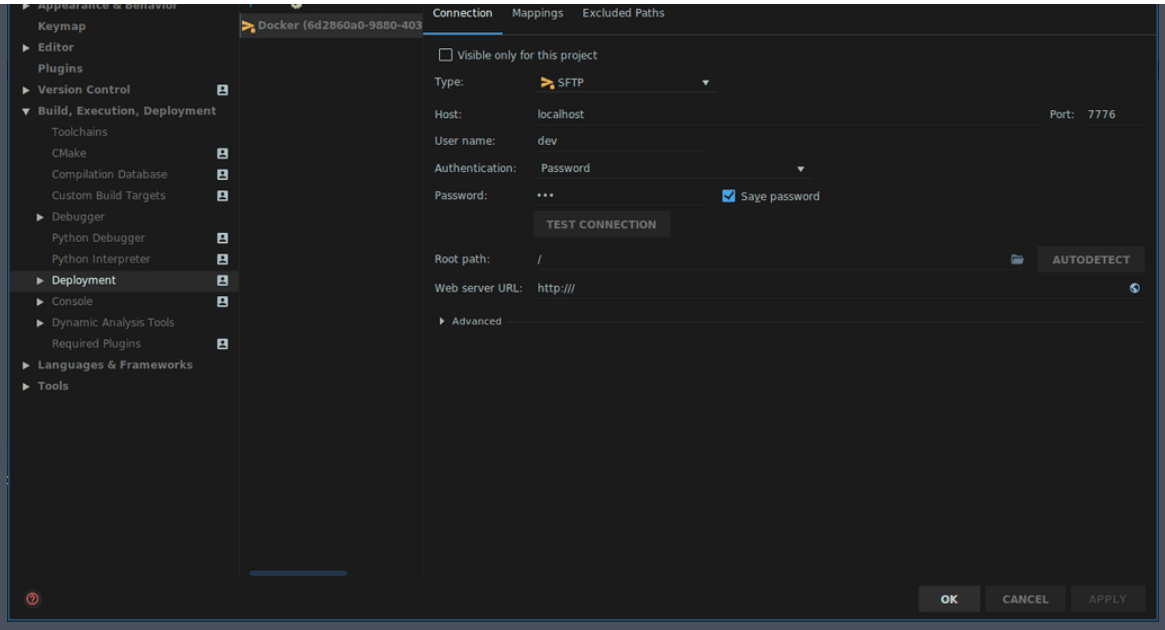


Go to **File -> Settings -> Build, Execution, Deployment -> CMake**. Add a new profile and name it **Debug-Remote**. Set the toolchain to our previously created **Docker** toolchain.

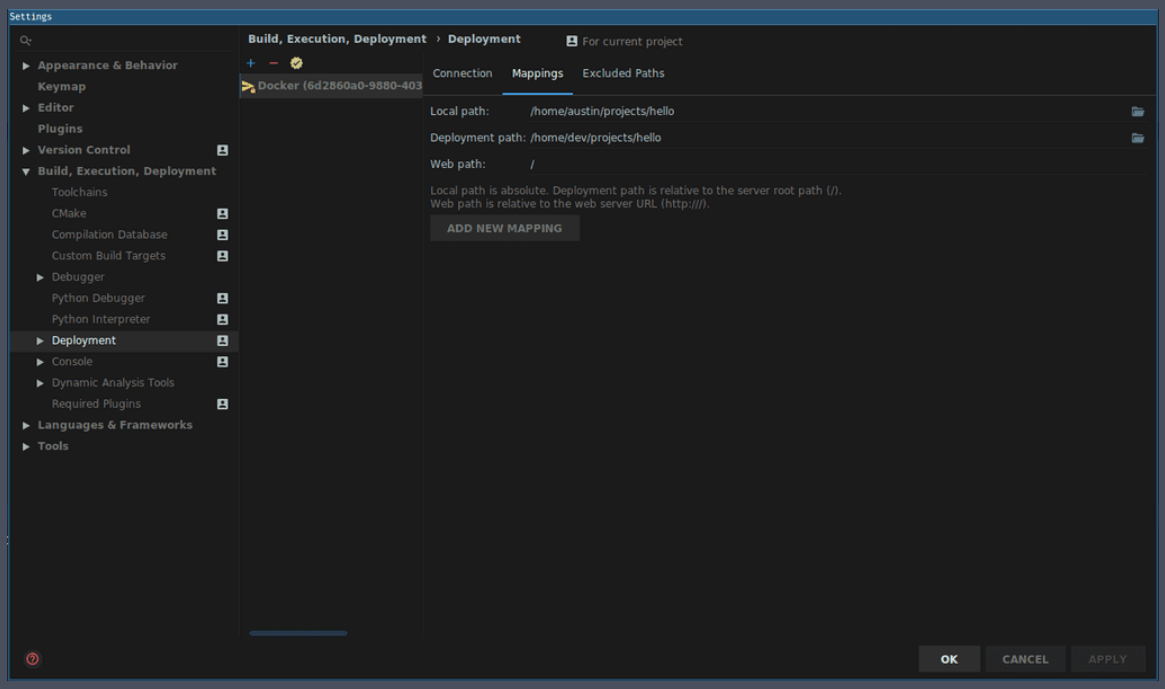


Go to **File -> Settings -> Build, Execution, Deployment -> Deployment**. Select the existing **Docker** entry and fill out the information if it isn't already there. Ensure the **Root path** is set to **/**.

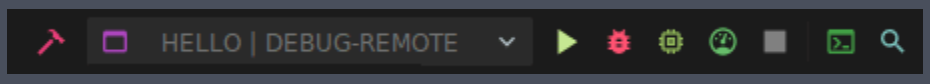




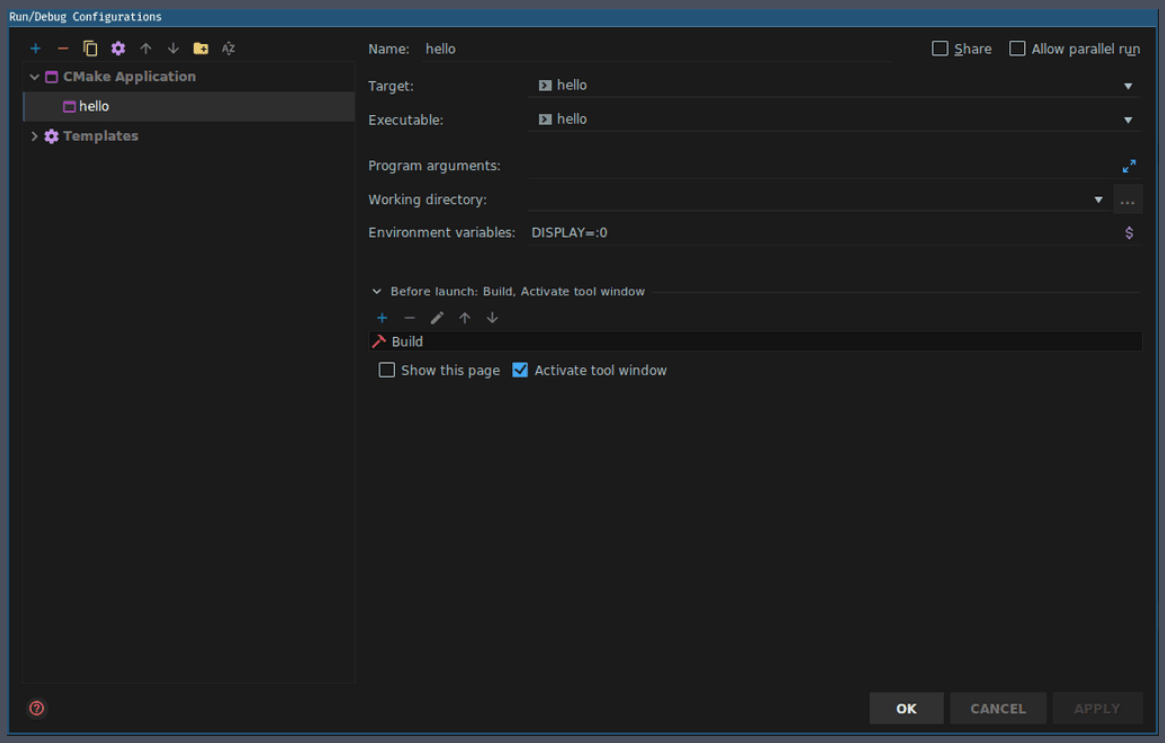
Go to the **Mappings** tab and set your local project directory to be mapped to a directory in the container.



Exit out of the settings and select the **Debug - Remote** profile from the dropdown in the upper right corner. If you don't see it there, make sure that you're able to SSH into the container normally (**ssh -p 7776 dev@localhost**).



We need to do two more things to ensure that we're able to run an X11 application in the container. In a shell on your host, run **echo $DISPLAY** and note the output. It's probably **:0** but could be something else. Then select **Edit Configurations** from the same build configuration dropdown in CLion and add **DISPLAY=:0** (or whatever the output was) to the list of environment variables. This connects the display of our host to the display of the container.

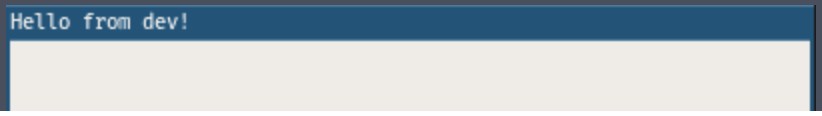


Finally, on your host, run **xhost +**, which allows the X Server to receive connections from clients outside of our host.

To build, right click on the name of your project and select **Reload CMake Project** which will sync the files to the container.

Now you should see that your syntax highlighting and code completion are working perfectly even though the headers are not on your host machine. Awesome.

Press **Shift F10** to build and run and you should get a Qt widget with a title showing a friendly message from the container.

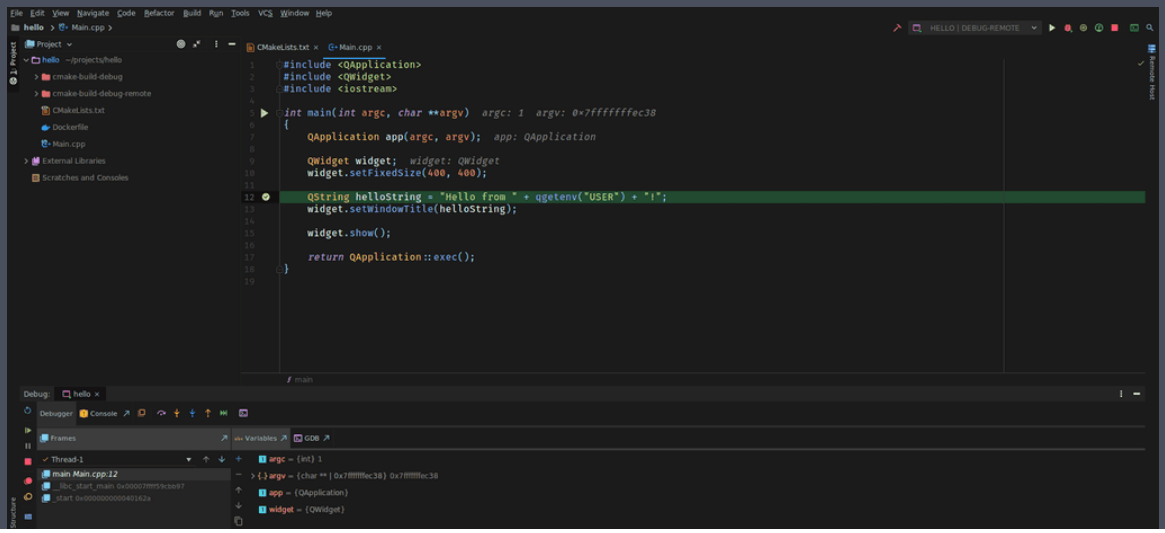


One last thing. If you want to debug an X11 application, you'll need a way to get the **DISPLAY** environment variable into GDB. The easiest way I've found is to create **~/.gdbinit** in the container with the following line:

```
1    set environment DISPLAY :0
```

GDB will load **.gdbinit** when it runs and will be good to go.

You can then set a breakpoint and press **Shift F9** and you will be able to debug a remote executable with all of the symbols present. Awesome.



Further Reading

- 1
- 2
- 3

Last Edited: Dec 20, 2022